IDEA

WHAT IS IT FOR

Il nostro progetto è pensato per tutti coloro che vogliono praticare sport e attività all'aperto sentendosi sicuri in ogni momento.

Grazie ai nostri braccialetti Aiotino si andrà a creare una sorta di **community di soccorso**, dove:

- ogni utente viene avvisato se sta entrando in una zona isolata (senza altri utenti)
- in caso di incidente gli utenti vicini vengono avvisati.

DESIGN

La **scelta** di realizzare proprio un braccialetto è legata al fatto che vogliamo un oggetto piccolo, leggero e comodo da portarsi con sé per praticare qualunque tipo di sport all'aperto (dalla corsa al tracking, fino al ciclismo). Essendo poi sul braccio è facile notare gli avvisi sotto forma di luci di colori differenti e suoni/vibrazioni.

Con Solidworks abbiamo realizzato un **modello visivo** del progetto. Il **prototipo** fisico realizzato è ovviamente differente, ma mantiene le stesse funzioni.

WHAT IT DOES/ MAIN FEATURES

Il nostro progetto ha due funzioni principali:

1. Raccoglie i dati gps degli utenti in cloud, e utilizzando questi insieme a un algoritmo di intelligenza artificiale avverte ogni utente del suo **stato di isolamento** attraverso dei led
2. Grazie a un accelerometro, riesce a rilevare eventuali cadute/colluttazioni degli utenti, e in caso di incidente avvisa attraverso un allarme/vibrazione gli utenti nelle vicinanze, che possono prestare soccorso seguendo le coordinate che compaiono su una mappa nei loro cellulari

THE PROTOTYPE

Il nostro prototipo è stato realizzato usando come **microcontrollore/board** un ESP32, come **sensori** un accelerometro e un modulo gps e come **attuatori** dei led e un cicalino

COST OF PARTS

Per quanto riguarda il costo totale del prototipo siamo attorno ai **20€**, anche meno se prendiamo i pezzi in grandi quantità.

ARCHITECTURE

COMMUNICATION DIAGRAM

Lo schema di comunicazione è il seguente:

- Il **microcontrollore,** nel nostro caso un esp32, raccoglie i dati dai **sensori**, inoltrandoli al **bridge**, e riceve altri dati dal **bridge**, che invia poi agli **attuatori.** La comunicazione bidirezionale con il bridge avviene tramite il protocollo BLE

- Il **bridge**, per noi un'app MIT sul cellulare, comunica inoltre bidirezionalmente con il **server** tramite il protocollo HTTP
- Infine, il **server,** implementato con Flask sul PC, gestisce tutta la comunicazione con DB, UI,algoritmo AI

SENSORS

Per quanto riguarda i sensori abbiamo usato:

- Come accelerometro per rilevare le cadute, **MPU6050,** che raccoglie i dati di accelerazione lungo 3 assi e usa il protocollo I2C per comunicare con il microcontrollore, il quale fa da master e raccoglie i dati tramite la libreria Wire.h di Aruduino
- Come modulo gps per raccogliere le posizioni degli utenti, **NEO6M,** che grazie a un'antenna di ceramica si connette ai satelliti gps e manda i dati al microcontrollore grazie alla comunicazione seriale. Il microcontrollore legge i dati sfruttando la libreria TinyGPS++ di Arduino.

COMMUNICATION PROTOCOLS

Nello specifico:

Il **protocollo I2C** è un protocollo asincrono che sfrutta il meccanismo master/slave per selezionare il ricevitore del messaggio. Nel nostro caso l'ESP32 fa da master e raccoglie i dati dell'accelerometro. Per implementare la comunicazione vanno connessi i pin SCL (di sincronizzazione) e SDA (dei dati) dell'accelerometro ai rispettivi nella board.

La comunicazione seriale tra gps e board si basa sul **protocollo asincrono UART,** dove in assenza di comunicazione il segnale = 1, se si vuole inviare un frame si deve inviare uno start bit =0, la lunghezza del messaggio, il messaggio (8 bit), e un end bit. Per implementarlo il pin TX (di trasmissione) del gps va connesso al pin RX (di ricezione) della board e viceversa.

ACTUATORS

Gli attuatori sono semplici:

- **BUZZER** per avvertire l'utente che un utente vicino ha avuto un incidente
- Dei **LED** per indicare lo stato di isolamento dell'utente e se la connessione BLE ESP32-bridge ha avuto successo

Il microcontrollore invia i dati agli attuatori semplicemente scrivendo sui pin i valori 0=LOW=OFF o 1=HIGH=ON ricevuti dal bridge

BLUETOOTH LOW ENERGY

Passiamo ora alla comunicazione bidirezionale tra ESP32 e BRIDGE, che avviene tramite BLE, una tecnologia wireless molto efficiente in termini di energia e quindi perfetta per il nostro braccialetto.

Lo schema della comunicazione è:

1. I dati da scambiare sono impacchettati in un SERVICE (univoco per ogni braccialetto), a sua volta composto da CHARACTERISTICS, identificate con degli UUID, che conterranno i dati gps e di accelerazione da inviare al bridge, e i dati di isolamento/allarme da ricevere.
2. Ogni braccialetto inizia una fase di SETUP e ADVERTISING, dove diventa individuabile dagli altri dispositivi.

3. Il bridge, su cui è stato registrato il mac address del braccialetto associato, dopo una fase di SETUP, effettua uno SCANNING automatico dei dispositivi BLE nelle vicinanze, e se trova il device associato, ci si connette e si registra per ricevere i messaggi. Comincia quindi lo scambio dati.

BRIDGE: MIT APP INVENTOR

Il bridge è implementato attraverso un'app nel cellulare, realizzata con MIT APP INVENTOR.

Si connette all'esp32 via BLE e al sever tramite richieste http

Più nello specifico:

- Dopo il login o signup, che avvengono con delle POST HTTP, il bridge si connette al **device associato** via BLE
- Riceve i dati di posizione, e accelerazione totale dal braccialetto associato
- Verifica se c'è stata una caduta (accelerazione supera una soglia), e inoltra i dati di posizione e caduta(0/1) e id del braccialetto al **web server** sottoforma di POST HTTP
- Riceve dal web server con delle GET HTTP tre variabili che inoltra al device:
  -isolato=0/1 (utente isolato)
  -predizione=0/1 (zona di solito isolata)
  -allarme=0/1 (qualcuno vicino è caduto)
  (oltre a delle varabili con la posizione della caduta)
- Nel caso in cui qualcuno nelle vicinanze cade e invia l'allarme, fa comparire nel device una mappa con le coordinate della caduta

MIT: COMMUNICATION WITH THE SERVER

Come già detto, la comunicazione con il server avviene sottoforma di **richieste HTTP**, dove i dati vengono scambiati in formato JSON.

Le richieste possono essere viste sfruttando l'interfaccia SWAGGER, andando all'indirizzo *web_server_url/docs* durante l'esecuzione del server, che rimanda alla pagina: *swagger.json*.

MIT: USER INTERFACE

La stessa app MIT svolge anche ruolo di INTERFACCIA UTENTE.

- Viene infatti utilizzata per effettuare il **login** di utenti già registrati e il **signup** di nuovi utenti.
- In caso di incidente ai braccialetti degli utenti vicini arriva una notifica sul cellulare e nell'app compare un pulsante, che se premuto rimanda alla mappa di **Google Maps**, che mostra le coordinate della caduta e mostra il percorso di come raggiungerlo.
- Un utente se cade ma sta bene e non ha bisogno di soccorso, può premere un **pulsante** specifico sull'app e gli altri utenti riceveranno un messaggio che lui sta bene.

WEB SERVER

Come web server per gestire le richieste e lo scambio dati abbiamo usato il framework **Flask.**

Esso è dotato inoltre di un'estensione, detta **SQLAlchemy,** molto comoda per interfacciarci con il **database**, nel quale il server salva i risultati delle **POST** del bridge: id, password, posizione, caduto

Grazie alle **GET** il server invia in ogni istante a ogni utente dei valori:

- Isolato=0/1 per indicare se c'è o meno qualche utente nelle vicinanze
- Predizione=0/1 per indicare se l'utente è in una zona che a quell'ora è solitamente popolata o no

Invia inoltre tre valori che di default sono settati a 0:

- Caduto, che se =1 indica che un utente vicino è caduto
- Latitudine
- Longitudine, che indicano la posizione corrente dell'utente che è caduto

Infine, il server si occupa di allenare e testare l'algoritmo **AI** di predizione del numero di persone in ogni zona, riportando i risultati nel database.

RELATIONAL DB

Più nello specifico,

SQLALCHEMY è uno strumento/libreria di **Object Relational Mapping**, che permette di interagire con il database tramite query in Python e non in SQL.

Il database utilizzato è **SQLITE**, perché il più leggero e facilmente implementabile (è un semplice file .db che viene creato e aggiornato a runtime).

Le tabelle presenti nel database sono:

- Utente= contiene id/password degli utenti registrati
- Braccialetto= per ogni braccialetto connesso ha i dati in tempo reale di posizione e caduta
- Presenza= per ogni zona d'interesse (nel nostro caso tre parchi di Modena) e per ogni ora della giornata contiene l'id dell'utente in quella zona.
- Predizione=contiene per ogni zona e ora della giornata il numero di persone predette dall'algoritmo AI

FB PROPHET

Passiamo quindi all'ultimo componente del nostro progetto: **FB PROPHET.**

È un algoritmo di AI realizzato da Facebook e usato per fare **predizioni di serie temporali** con trend annuali, settimanali o nel nostro caso giornalieri. È inoltre robusto ai missing values. Noi l'abbiamo usato per prevedere quanto alcune zone di interesse (parco Ferrari, Amendola, Resistenza) sono isolate/popolate ad ogni ora.

Il modello viene allenato usando un file .csv, contenente la storia degli eventi passati con una certa temporalità.

Quindi:

1. Creiamo e aggiorniamo giornalmente il nostro .csv, prendendo i dati della tabella "Presenza" e usandoli per contare il numero di persone presenti a ogni ora di una certa giornata in ognuna delle zone d'interesse.
2. Alleniamo e testiamo giornalmente FBProphet, passandogli il csv aggiornato. Esso prevedrà il numero di persone presenti a ogni ora del giorno successivo (o intervallo futuro) per ogni zona.
3. Riportiamo le predizioni nella tabella "Predizione" del db. Il server userà i dati di questa tabella per avvertire gli utenti del loro stato di isolamento.

LEDS AND ISOLATION STATES

Ora abbiamo tutte le parti per capire come funzionano gli stati di isolamento indicati con i led.

Dai dati di posizione ogni braccialetto riceve una variabile ISOLATO=0/1

Dai dati di predizione di Prophet ogni braccialetto riceve una variabile PREDIZIONE=0/1

➔ Se l'utente non è isolato (ISOLATO=0), non si accendono led nel braccialetto

➔ Se l'utente è isolato ma la zona è solitamente popolata (ISOLATO=1, PREDIZIONE=0), si accende il led giallo
➔ Se l'utente è isolato e la zona è isolata (ISOLATO=1, PREDIZIONE=1), si accende il led rosso

POSSIBLE IMPROVEMENTS

Ovviamente il nostro è solo un prototipo, che in futuro dovrà essere migliorato:

- Aggiungendo un algoritmo di ML che migliori il **riconoscimento delle cadute** (con solo l'accelerometro uno strattone/movimento brusco può innescare l'allarme
- Costruendo una **mappa di rischio**, che indica a ogni ora le zone da evitare, consultabile dall'app che tenga conto delle zone in cui la gente cade spesso, la condizione del terreno e il meteo
- Aumentando il **numero di zone** d'interesse su cui fare predizioni (noi per semplicità ne abbiamo usate tre)
- Lavorando sulla **scalabilità**:
  - Inserire meccanismo di associazione automatico tra app e braccialetto (noi dobbiamo inserire manualmente il mac address del device associato)
  - Aumentare la privacy e la sicurezza dei dati sensibili e gps
  - Usare un database più flessibile e complesso per inserire molti dati senza perdere di performance

DEMO

Abbiamo testato due situazioni:

❖ Utente è isolato in una zona che solitamente a quell'orario è prevista essere popolata -> led giallo
❖ Utente è isolato in una zona isolata -> led rosso, ma poi si collega un altro utente->led spenti.
Il nuovo utente cade->parte cicalino e compare pulsante mappe